The Ins and Outs of Optimizing Compilers

by Jeff Stefan

Want to write better code? Consider understanding compiler optimizations.

If you're not writing computer programs in hand-coded assembly, chances are you're using a compiler. If you're using a compiler, chances are the compiler is performing optimizations on your code, possibly unbeknownst to you. Compilers are where text touches silicon; your abstract source code becomes a concrete program that runs on an actual machine. Being aware of compiler optimization techniques can save you a measured amount of frustration, especially when debugging your code.

Fundamental Compiler Technology

Compilers are usually broken up into two parts: a front end and back end. The front end takes care of scanning and parsing the source code, and the back end takes care of generating and optimizing code. Figure 1 illustrates the basic components of a compiler.

As the source code enters the compiler, the front end reads the characters in the input stream and separates them into tokens, as shown in Figure 2. This is the part of the compilation process that's called lexical analysis. Tokens are symbols that cannot be reduced any further. Tokens are not restricted to single characters. For example, if you have a variable named MyVar in your source program, MyVar cannot be reduced any further, therefore, MyVar is a token. In some compiler literature, tokens are occasionally called atoms.

As the tokens are scanned, their type is determined and the token and its type are placed in a data construct called a symbol table.. The symbol table is a convenient storage device that allows other components to access expressions, variables, and types. Symbol tables are usually implemented as linked lists.

The next component in the compiler's front end is the parser. The parser does essentially two things. First, it determines whether the input represents a legal construct of the language. Second, it puts the input expression into an intermediate form that makes it easier for the code generator to process. The parser tries to mach the sequence of tokens to a set of rules called productions. The set of productions that make up the rules for a language is called a grammar.

If you look in Appendix A, Section A13 of Kerninghan and Richie's book *The C Programming Language*, the grammar for C is listed. Productions or grammar rules are usually expressed in Backus-Normal Form, or BNF. BNF was invented and used to express the grammar for the first implementation of FORTRAN. BNF is an adaptation of the way linguists express natural language. John Backus is credited for creating FORTRAN and for first developing BNF, so BNF is named after him. BNF is a convenient way to express a programming language's grammar and has stood the test of time. An example of a grammar for simple arithmetic expressions is shown in Figure 3.

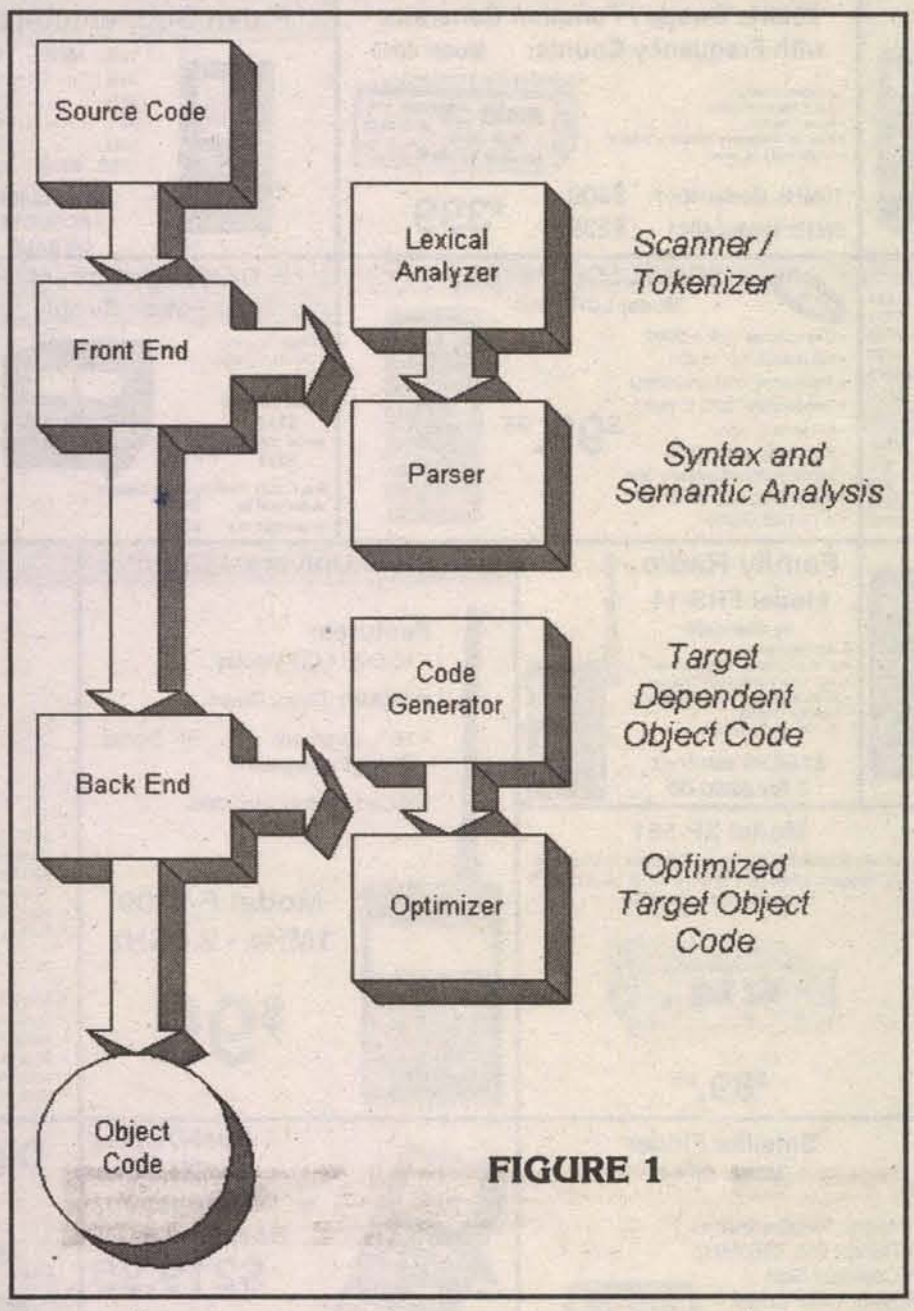

FIGURE 1

Suppose the parser encounters the input expression i = j + k * z. In order to determine if this is indeed a legal expression, the parser needs to match it to the grammar rules given in Figure 3. Is i = j + k * z a legal expression? First, the parser must determine whether j + k * z can be derived from the grammar rules. If it can, then code can be generated for this expression, and the resulting computation can be stored in the variable i.

Here's the derivation:

First, we can consider j + k * z to be an expression <Expr>. From the grammar rules, <Expr> can be rewritten as <Expr> ::= <Expr> * <ID>. Since <ID> can be rewritten as a letter or a digit, this derivation resolves to <Expr> * z. Next we need to figureure j + k into the derivation. We do this by rewriting <Expr>* z with <Expr> + <ID> * z. We can substitute the token k for <ID>, resulting in <Expr> + k * z. Lastly, we can rewrite <Expr> as <ID>, yielding <ID> + k * z. We can substitute the token j for <ID> , resulting in j + k * z. Since we were able to match the expression to the grammar rules, the input expression is indeed legal and can be correctly compiled. Figure 4 shows the complete derivation sequence.

Code Generation

After parsing is complete, the grammar rules are matched, and the symbol tables filled, code generation can begin. Code generation may take the form of assembly language, or the compiler may skip assembly code generation and directly produce code for the target machine. This is called native code generation. Most C compilers for embedded systems produce assembly code, which is then optimized before any object code is produced.

Low-Tech Optimization

There's a lot you can do before your source code is ever compiled. Careful programming can make the compiler's job easier. By using a few hand-optimization techniques, your code will run tighter and faster when it's compiled. Hand optimization is a low-tech approach, but it pays off. For example, look at the following C code fragment:

```
for (i = 0; i < 10; i++)
{
  x = 10;
  y = i + x;
}
```

Simply removing the assignment x = 10 rad-

ically increases the efficiency of this code fragment. This type of source level optimization is fairly obvious, and even beginning programmers recognize this. More subtle methods include using pointers instead of using array indices as the following example illustrates:

```
/*
* Non-pointer method of access buffer elements
using an index variable
*/
 printf("\nBuffer access using index variable\n");
for(I = 0; i < 10; i++)
 printf("%c",Buffer[i]);
```

Performance of this code is increased by using a pointer:

```
/*
* Set pointer to start of Buffer and print values
*/
printf("\nBuffer access using pointer\n");
ptr = Buffer;
while(*ptr)
 printf("%c",*ptr++);
printf("\n");
```

These are simple optimizations that a careful programmer can accomplish on his or her own.

Profiling

Another useful procedure is to profile a program without any compiler optimizations turned on. Profilers are important, but sometimes little used tools. Profilers — or performance analyzers — find run-time bottlenecks in your program while it executes. You compile and link your program, then execute it under the profiler's control. When the program terminates, the profiler generates program performance statistics.

You can discover where your program spends most of its time, how many times certain routines were called, how much CPU time was consumed, and how many interrupts occurred, among other things. Profilers are extremely powerful tools that are sometimes bundled for free with some compilers.

Common Optimization Techniques

A compiler doesn't necessarily produce object code or an executable program exactly the way your program intended it to. All a compiler is expected to do is to reproduce the way the program was intended to behave.

From a compiler's perspective, as long as the program produced works the same way as your program text intended, what happens algorithmically within the program doesn't matter. This is surprising, but true. Knowing that the optimized program a compiler produces may not be exactly what you programmed is a good thing to bear in mind when you're debugging your code.

Some of the compiler optimization methods have strange sounding names that, on the surface, don't seem to make a lot of sense, such as loop induction, peephole optimization, or constant folding. It's worth the time to understand optimization terminology so you can make informed decisions about what optimizations to allow when compiling your code.

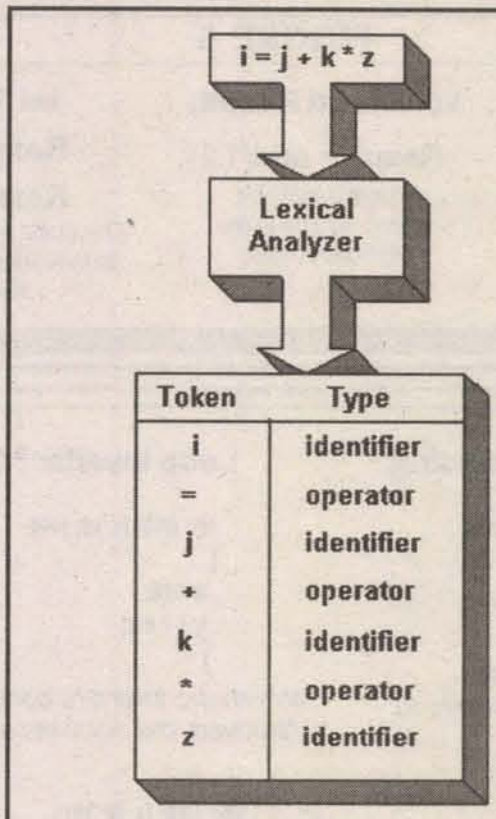

FIGURE 2

```
<Expr>   ::=   <ID>   |
                <Expr> +  <ID>   |
                <Expr> *  <ID>
 <ID>    ::=  <letter>  |  <ID> <letter>  |
                <ID> <digit>
<letter> ::=  A|B|C|...|a|b|c|...z
<digit>  ::=  0|1|...9
```

FIGURE 3

FIGURE 4

```
Rule Matching for  j + k * z
 <Expr> ::= <Expr> * <ID>
 <Expr> ::= <Expr> * z
<Expr> ::= <Expr> + <ID> * z
<Expr> ::= <Expr> + k * z
<Expr> ::= <ID> + k * z
<Expr> ::= <ID> + k * z
<Expr> ::= j + k * z
```

Register Allocation

Register allocation is probably the simplest and most ubiquitous of all optimizations. When a variable is accessed, the compiler first looks if it can use a CPU register to store the variable instead of storing it in external memory. This is a speed-oriented optimization, since no memory address bus cycles are used. Register allocation must be done in the back end of the compiler during code generation, since the front end of the compiler does not know how many registers the target machine contains.

Dead Code Elimination

If a value is computed but never used, it's dead code. An optimizing compiler will recognize this and eliminate it in the code generation phase. This can be distracting if code stubs are placed in a program to be filled in later. For example, if you have the following lines in your program:

```
int Result;
Result = add(1,2);
```

and your program never used the variable Result again, the code may be completely optimized out of your program. If this was a program stub that you were going to build upon later, you'd be in trouble. A way to place stubs into the code without having them optimized out and without turning off compiler optimizations is to add the following assignment immediately after the call to add(1,2):

```
Result = Result;
```

This uses the variable Result without really doing anything. This simple trick costs nothing and allows the stub to avoid the optimizer's axe. You could also declare Result as volatile. Declaring a variable as volatile forces the compiler to use the variable under all circumstances, since the keyword volatile suppresses any compiler optimizations. Be careful using the keyword volatile, since all of the variables of this type will never be optimized. See Figure 5 for a simple side-by-side comparison of this code.

Loops

Loops are obvious candidates for optimization. Most programs spend most of their CPU cycles within loops. Here are some common loop optimizations.

Loop Unwinding

Loop unwinding is a classic example of the code-size, code-speed tradeoff. When loop unwinding occurs, the loop is decomposed into linear code. The linear code is copied the number of times the code iterates. This produces more code, but it runs faster. An example of loop unwinding is shown in Figure 6.

Loop-Invariant Code

If code exists in a loop that doesn't change, then there is no reason for the code to be executed over and over again within the loop. It wastes CPU and bus cycles, which wastes time. Optimizing compilers that are sensitive to loop-invariance recognize this type of code and extract it from the loop. The loop-invariant code is placed above the loop, yielding more efficient code. We saw this example before, in the Low-

Tech Optimization section. An example of loop-invariant code elimination is show in Figure 6.

FIGURE 5

int Result;	volatile int Result;	int Result;
Result = add(1,2);	Result = add(1,2);	Result = add(1,2);
This code is optimized out by the compiler	*Result is NEVER optimized due to the keyword volatile*	Result = Result; *This code is not optimized out due to the simple benign assignment Result = Result;*

Loop induction

Pointers are more efficient than indexes, and addition is more efficient than multiplication. With loop induction optimization, the compiler attempts to replace array index multiplications with pointer additions, again yielding more efficient code. We saw a simple form of this in the Low-Tech Optimization section where we substituted pointers for array indexes in the for loop code example.

Constant Folding

This is a somewhat strange name for performing calculations at compile-time instead of run-time. When a compiler encounters expressions that can be evaluated at compile time, the compiler does the actual calculation and then stores the result. For example, a code fragment such as:

```
#define MAX_TEMPERATURE 100
y = 25;
x + y + 2 * MAX_TEMPERATURE;
```

is optimized to x + 225 by the compiler by utilizing constant folding.

Loop Unwinding

```
for (i=0;i<3;i++)
{
 x = x+i;
}
```

When this code unwinds, it resolves to:

```
x = x+0;
x = x + 1;
x = x + 2;
```

Loop Invariant Code

```
for (i=0; i< 10; i++)
{
 x =10;
 y = i + x;
}
```

When the invariant code is removed, the code resolves to:

```
x = 10;
for (i=0; i< 10; i++)
{
 y = i + x;
}
```

FIGURE 6

Peephole Optimization

The last technique we'll examine is peephole optimization. This is also sometimes called linear optimization. Peephole optimization is common throughout almost all compilers, since it is one of the easier techniques to implement. A peephole optimizer scans through the code examining only a small fragment at a time, like it's examining the code through a little peephole (hence the name).

The peephole optimizer then tries to improve the current fragment, possibly reducing redundant instructions or placing memory references into registers. After it's done optimizing a segment, the peephole optimizer examines another segment of code and continues optimizing.

Burned by the Optimizer

Optimizing compilers can sometimes cause confusion and cost time, which translates directly to money. Here's a personal experience I had with an optimizing compiler for a 32 bit processor that was used in an embedded system. We compiled and loaded a small test program that was supplied with the operating system code.

Since this was our first experience with this particular processor and operating system, we figured it was a good place to start. The program ran correctly on the target system, so we decided to add a new function. We declared a simple integer function that added two numbers together:

```
int add( int x, int y);
```

The actual call to the function in the body of the code was:

```
Result = add(1,2);
```

It was hard to think of a simpler function to start experimenting with. We compiled, linked, downloaded, and ran the code through the debugger on the target system. The debugger stepped right over the function call as if it didn't exist in the code! We checked the code, recompiled, downloaded, and ran again with the same result.

Everything appeared to be completely legal. We tried it on a PC with another compiler, and the line containing the function call was executed. We returned to the target environment, recompiled, and ran again. The debugger still stubbornly refused to execute the function call.

To add to our confusion, we could set a breakpoint within the function body, but could not set a breakpoint at the function invocation. Meanwhile, several hours had elapsed.

We decided to call the development environment vendor, thinking that we had encountered a bug in the compiler or debugger. After talking to a couple of support engineers, we finally happened upon one who had previously encountered a similar problem. His response was "Welcome to the world of optimizing compilers!" It turned out that our result value was never used in the program, so via dead code elimination, the compiler decided to optimize out the code.

The moral of this story is, however valuable compiler optimizations may be, TURN THEM OFF when you are initially developing code for a new system using new tools. Incrementally turn on optimizations as your code development progresses. If you don't, you may get burned when you're debugging. **NV**

BUY • SELL • CONSIGN

WANTED
- Electronic Test Equipment
- Electronics Inventory
- Scientific Equipment
- Mfg. Shop Equip.
- Data/Telco
- Memory

DEXIS
9749 Hamilton Road
Eden Prairie, MN 55344

We Sell Nationwide!
Call our Fax-On-Demand System, or our Internet site for current availability.
Visit our 25,000 sq. ft.showroom warehouse
M-F 9 to 6; Sat 9 to 5

PHONE: (612) 944-7670
INTERNET: WWW.GO2DEXIS.COM
FAX: (612)942-9712 • FAX-ON-DEMAND: 1-800-547-4636 X 300

Write In 107 on Reader Service Card.

PRINTED CIRCUIT BOARDS

QUALITY PRODUCT
FAST DELEVERY
COMPETIVE PRICING

PROTOTYPE THROUGH PRODUCTION

- * UL approved
- * Single & Double sided
- * Multilayers to 8 layer
- * SMOBC
- * LPI mask
- * Through hole or SMT
- * Nickel & Gold Plating
- * Routing or scoring
- * Electrial Testing
- * Artwork or CAD data
- * *Fast* quotes

PULSAR, INC
9901 W. Pacific Ave.
Franklin Park, IL 60131
Phone 847.233.0012
Fax 847.233.0013
Modem 847.233.0014
www.pulsar-inc.com

Write In 108 on Reader Service Card.